Chapitre 3: Optimisation exacte

INF889B — Algorithmes d'optimisation combinatoire

Alexandre Blondin Massé

Université du Québec à Montréal

Hiver 2020

Plan

1. Généralités
2. Programmation dynamique
3. Séparation et évaluation progressive
4. Paramètre fixe

Généralités

Type de problème

- Avant de considérer une solution **approchée**
- S'assurer que le problème est **NP-difficile**
- Certains problèmes en apparence difficiles peuvent être résolus en temps **polynomial**

→ programmation dynamique

→ mémoïsation

Problème NP-difficile

- Et même si le problème est NP-difficile
- Plusieurs **stratégies exactes**:

→ algorithme naïf

→ restriction de classe

→ séparation et évaluation progressive

→ à paramètre fixe, etc.

Restriction de classe

Idée de base

- Les instances qui nous intéressent ont des **propriétés particulières**
- On peut donc restreindre l'**espace des instances**

Exemples

- Graphes simples $\rightarrow$ graphes simples **planaires**
- Graphes simples $\rightarrow$ graphes d'**intervalles**
- Problème SAT $\rightarrow$ problème **2-SAT**
- Problème du sac à dos $\rightarrow$ instances à **valeurs entières**

Remarque: Une restriction ne garantit pas une diminution de difficulté

Approches non exactes

Deux **types** d'algorithmes

Algorithme approximatif

- On accepte d'avoir une **solution non optimale**
- Avec une **garantie** que l'**erreur** n'est pas trop grande
- **Avantage**: garantie théorique
- **Inconvénient**: souvent difficile à démontrer

Méta-heuristiques

- On accepte d'avoir une **solution non optimale**
- **Aucune garantie** sur la marge d'erreur
- **Avantage**: on peut obtenir de meilleures performances
- **Inconvénient**: idée très imprécise de la marge d'erreur

Programmation dynamique

Idée générale

Contexte

- On souhaite résoudre un problème d'**optimisation**
- Ce problème peut être exprimé de façon **récursive**
- Autrement dit, il peut être divisé en **sous-problèmes**
- Mais l'algorithme naïf utilise un temps **exponentiel**
- Car il effectue les **mêmes calculs** plusieurs fois

Solution

- On introduit un **tableau auxiliaire**
- Peut être de dimension **quelconque**
- On calcule les valeurs du tableau dans le **bon ordre**
- Puis on retourne la valeur **finale**
- Parfois on doit **reconstruire la solution** à partir du tableau
- Ou on peut conserver seulement une **partie** du tableau

$\rightarrow$ économise de la mémoire

Suite de Fibonacci

- La suite $\{f(n)\}_{n\geq 0}$ de Fibonacci est définie par

$$f(0) = 1, \quad f(1) = 1$$

$$f(n) = f(n-1) + f(n-2), \text{pour } n \geq 2$$

- Algorithme naïf:

```
1: fonction FIBO(n : naturel) : naturel
2:     si n ≤ 1 alors
3:         retourner n
4:     sinon
5:         retourner FIBO(n − 1) + FIBO(n − 2)
```

- **Problème?** Inefficace, car recalcule plusieurs fois le même nombre

Mémoïsation

- On enregistre le résultat si l'**entrée** a déjà été vue
- La fonction doit être **pure**
- De façon générale, on utilise un **tableau associatif**
- Ou un **tableau** directement si le domaine est de la forme $\{0, 1, \ldots, n-1\}$
- Dans l'exemple de FIBO, on initialise un tableau avec des valeurs spéciales, par exemple $\mathit{fibo}[n] \leftarrow \mathit{undefined}$

```
1: fonction FIBO(n : naturel) : naturel
2:     si fibo[n] = undefined alors
3:         si n ≤ 1 alors
4:             fibo[n] ← n
5:         sinon
6:             fibo[n] ← FIBO(n − 1) + FIBO(n − 2)
7:     retourner fibo[n]
```

Programmation dynamique

- Aussi appelée **programmation tabulaire**

$\rightarrow$ car elle consiste à remplir des tables

- Essentiellement une version **non récursive** de la mémoïsation

1: **fonction** FIBO(n : naturel) : naturel
2: $\quad$ *fibo*$[0] \leftarrow 1$
3: $\quad$ *fibo*$[1] \leftarrow 1$
4: $\quad$ **pour** $i \in \{2, 3, \ldots, n\}$ **faire**
5: $\quad\quad$ *fibo*$[n] \leftarrow$ *fibo*$[n-1]$ + *fibo*$[n-2]$
6: $\quad$ **retourner** *fibo*$[n]$

- **Remarque**: il n'est pas nécessaire de retenir toutes les valeurs, seules les *2 dernières* suffisent
- En pratique, plus **efficace** que la mémoïsation, car n'utilise pas la pile système (appels récursifs)

Coupe d'une tige

- Considérez une tige de longueur n
- On doit la découper en un nombre arbitraire de petites tiges
- Pour $i = 1, 2, \ldots, n$, soit v_i la valeur d'une tige de longueur i
- On souhaite **maximiser** la valeur totale
- Par exemple, supposons que $n = 8$ et que les valeurs sont données par le tableau suivant:

i	1	2	3	4	5	6	7	8
v_i	2	3	3	5	8	8	9	11

Questions

1. Proposez un algorithme naïf pour résoudre ce problème?
2. Quelle est sa complexité?
3. Pour $i = 1, 2, \ldots, n$, soit $V[i]$ la valeur maximale qu'on peut obtenir en coupant une tige de longueur i. Donnez une équation récursive pour $V[i]$ en fonction de valeurs de la forme $V[i']$, où $i' < i$.
4. Déduire de l'équation précédente un algorithme de programmation dynamique qui permet de résoudre le problème
5. Quelle est sa complexité?

Marche dans une matrice

- Considérez une matrice de dimensions $m \times n$, par exemple

$$\begin{bmatrix} 4 & 7 & 8 & 6 & 4 & 5 \\ 1 & 3 & 5 & 9 & 2 & 4 \\ 3 & 7 & 4 & 4 & 8 & 2 \\ 9 & 2 & 8 & 7 & 2 & 4 \end{bmatrix}$$

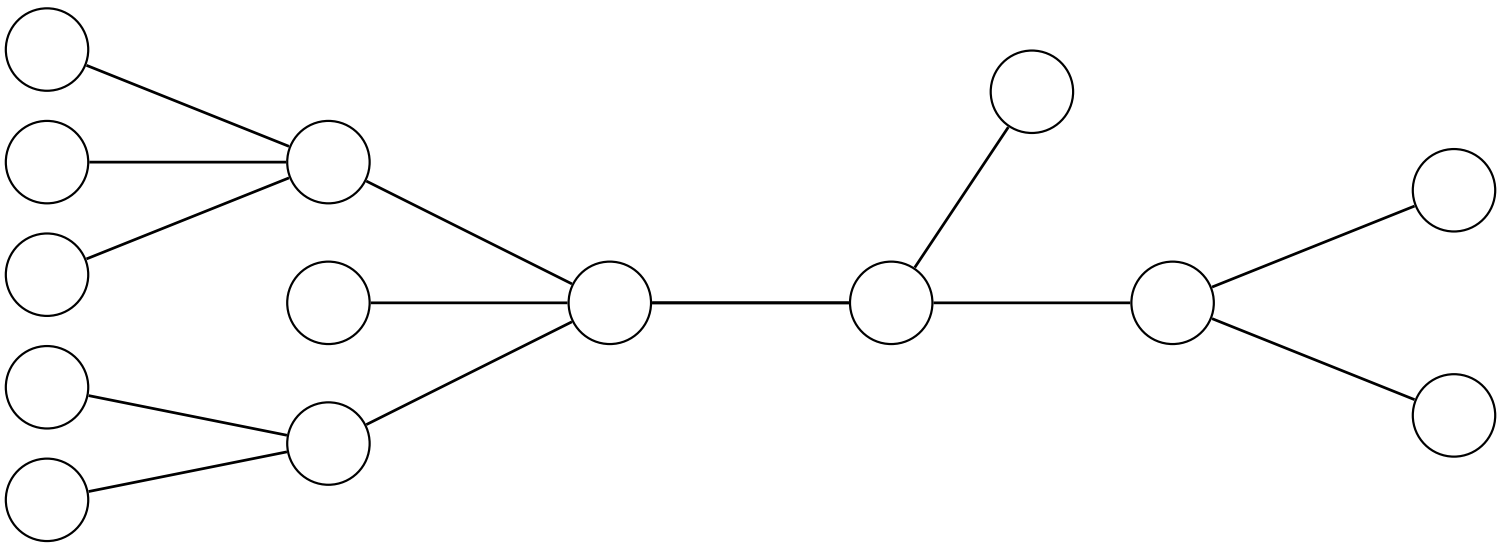
- On commence dans le coin **supérieur gauche**
- On souhaite se rendre dans le coin **inférieur droit**
- Les seuls déplacements permis sont *est* et *sud*
- On aimerait **minimiser** le coût total

Questions

1. Proposez un algorithme naïf pour résoudre ce problème?
2. Quelle est sa complexité?
3. Pour $i = 1, 2, \ldots, m$ et $j = 1, 2, \ldots, n$, soit $C[i, j]$ le coût minimal pour se rendre de l'entrée $(1, 1)$ à l'entrée (i, j). Donnez une équation récursive pour $C[i, j]$ en fonction de valeurs de la forme $C[i', j']$, où $i' + j' < i + j$.
4. Déduire de l'équation précédente un algorithme de programmation dynamique qui permet de résoudre le problème
5. Quelle est sa complexité?

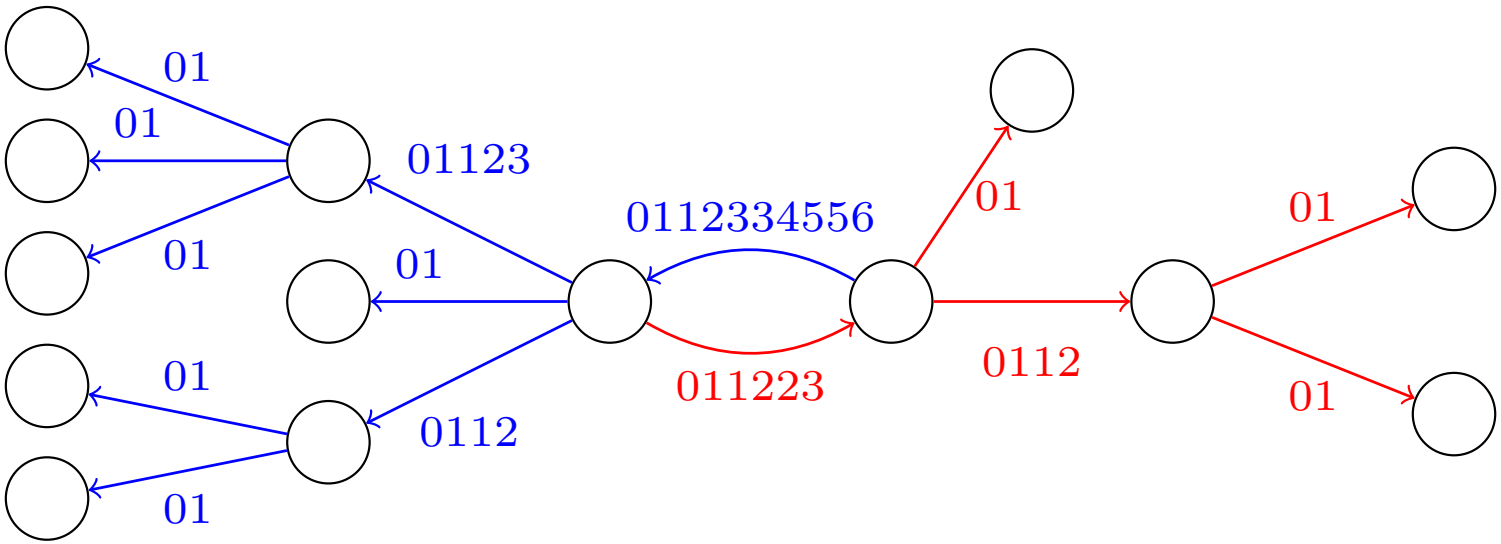
Nombre de feuilles d'un sous-arbre d'un arbre

- Soit T un arbre non orienté
- Pour tout naturel i, on cherche le nombre **maximum de feuilles** qu'on peut réaliser avec un sous-arbre de T de taille i
- **Exemple**:



Solution par programmation dynamique

- Voir la section 4 de l'article Fully leafed induced subtrees, par Blondin Massé, de Carufel, Goupil, Lapointe, Nadeau et Vandomme
- L'idée consiste à stocker dans chaque **orientation** d'arête le plus grand nombre de feuilles qu'on peut réaliser avec un sous-arbre de taille i



Séparation et évaluation progressive

Idée de base

- Même idée que l'algorithme **naïf**
- Mais on structure l'espace de façon **hiérarchique** (arborescence)
- À chaque noeud, on vérifie s'il est **prometteur** (potentiel de trouver une solution optimale)

$\rightarrow$ Si **oui**, on poursuit l'exploration

$\rightarrow$ Si **non**, on abandonne la sous-arborescence

Remarques

- **Plus efficace** que l'algorithme naïf
- L'espace exploré est **plus petit**
- **Plus de travail** à implémenter
- Souvent **exponentiel** quand même
- Évaluation du **potentiel** d'un noeud doit être peu coûteuse

$\rightarrow$ Voir cours **INF7440 Conception et analyse des algorithmes**

Exemple: problème du sac à dos

- On considère n objets, numérotés 1, 2, ..., n
- Chaque objet a une **valeur** v_i
- Et un poids w_i
- La capacité maximale du sac est W
- On souhaite maximiser la valeur totale en respectant la capacité, c'est-à-dire

$$\begin{aligned} \text{maximiser} \quad & \mathbf{v} \cdot \mathbf{x} \\ \text{sous les contraintes} \quad & \mathbf{w} \cdot \mathbf{x} \leq W \\ & \mathbf{x} \in \{0,1\}^n \end{aligned}$$

- Ce problème est **NP-difficile**

Note

- Le problème est polynomial si $v_i, w_i \in \mathbb{N}$
- On utilise la **programmation dynamique** pour cela

Solution naïve (1/2)

- Prenons le cas où $W = 5$ et où les valeurs et poids sont donnés par le tableau suivant:

i	1	2	3
v_i	3	4	3
w_i	3	3	2

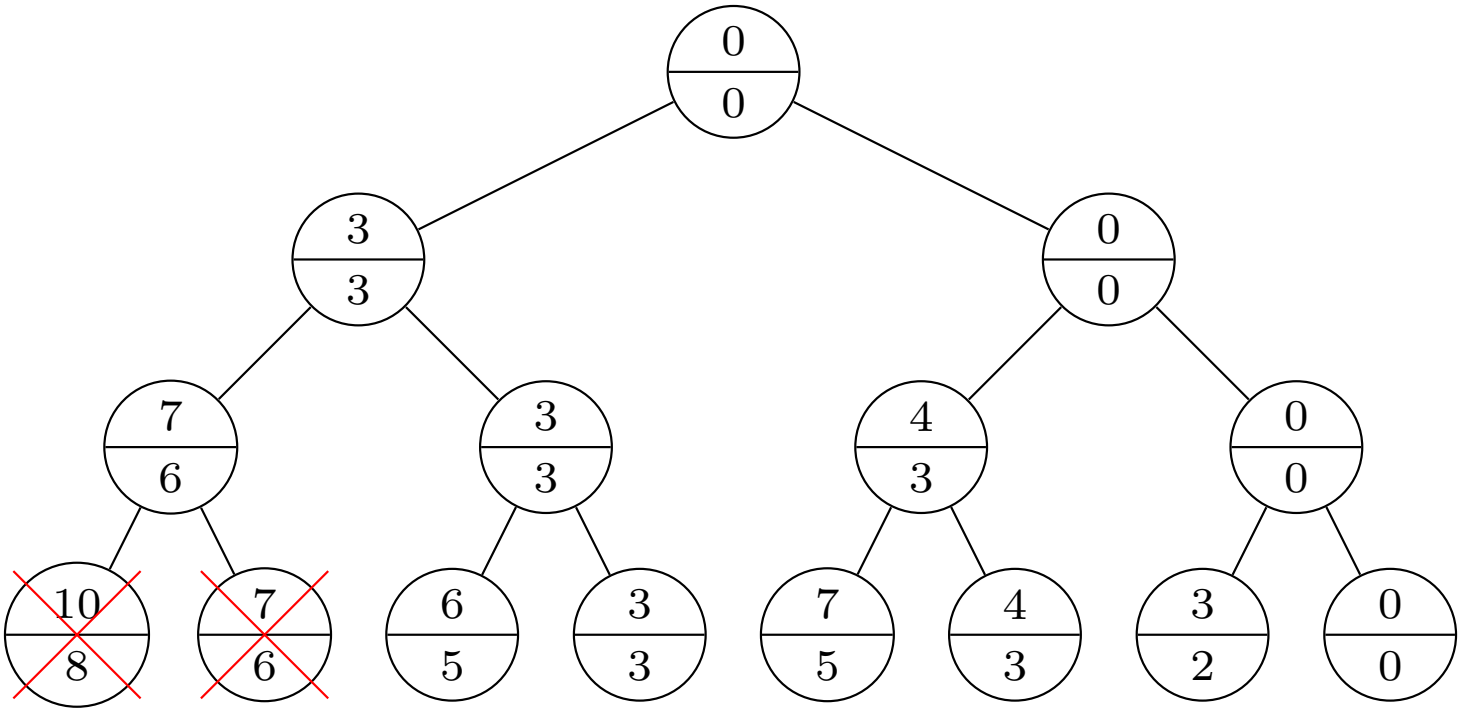
- L'**arbre** de tous les cas possibles est le suivant:

Solution naïve (1/2)

- Prenons le cas où $W = 5$ et où les valeurs et poids sont donnés par le tableau suivant:

i	1	2	3
v_i	3	4	3
w_i	3	3	2

- L'**arbre** de tous les cas possibles est le suivant:



Solution naïve (2/2)

```
1:  fonction SacADosNaif : réel
2:      x_meilleur ← 0
3:      SacADosNaif(0, 0)
4:      retourner x_meilleur
5:
6:  fonction SacADosNaif(i : entier, x : vecteur booléen) : réel
7:      si i ≤ n alors
8:          x' ← x
9:          x'[i] ← 1
10:         SacADosNaif(i + 1, x')
11:         x'' ← x
12:         x''[i] ← 0
13:         SacADosNaif(i + 1, x'')
14:     sinon si x · w ≤ W et Σx > Σx_meilleur alors
15:         x_meilleur ← x
```

Par séparation et évaluation progressive

1: **fonction** SacADos : réel
2: $\quad \mathbf{x}_{meilleur} \leftarrow \mathbf{0}$
3: $\quad v_{meilleur} \leftarrow 0$
4: $\quad$ SacADos$(0, \mathbf{0})$
5: $\quad$ **retourner** $\mathbf{x}_{meilleur}$
6:
7: **fonction** SacADos$(i$: entier, $\mathbf{x}$: vecteur booléen$)$: réel
8: $\quad w_{courant} \leftarrow \mathbf{x} \cdot \mathbf{w}$
9: $\quad v_{courante} \leftarrow \mathbf{x} \cdot \mathbf{v}$
10: $\quad v_{fractionnaire} \leftarrow$ SacADosFractionnaire$(V[i,n], W[i,n], W - w_{courant})$
11: $\quad$ *valide* $\leftarrow w_{courant} \leq W$
12: $\quad$ *prometteur* $\leftarrow v_{fractionnaire} + v_{courante} > v_{meilleur}$
13: $\quad$ **si** $i \leq n$ **et** *valide* **et** *prometteur* **alors**
14: $\quad\quad \mathbf{x}' \leftarrow \mathbf{x}$
15: $\quad\quad \mathbf{x}'[i] \leftarrow 1$
16: $\quad\quad$ SacADos$(i+1, \mathbf{x}')$
17: $\quad\quad \mathbf{x}'' \leftarrow \mathbf{x}$
18: $\quad\quad \mathbf{x}''[i] \leftarrow 0$
19: $\quad\quad$ SacADos$(i+1, \mathbf{x}'')$
20: $\quad$ **sinon si** *valide* **et** $v_{courante} > \sum v_{meilleure}$ **alors**
21: $\quad\quad \mathbf{x}_{meilleur} \leftarrow \mathbf{x}$
22: $\quad\quad v_{meilleur} \leftarrow \mathbf{x}_{meilleur} \cdot \mathbf{v}$

Paramètre fixe

Idée de base

- Cas particulier de la **restriction de classe**
- On suppose qu'un ou plusieurs paramètres sont **constants** ou **bornés**

$\rightarrow$ Degré maximum d'un graphe
$\rightarrow$ Largeur arborescente (*treewidth*) d'un graphe
$\rightarrow$ Largeur de chemin (*pathwidth*) d'un graphe
$\rightarrow$ Nombre de machines (ressources) disponibles

- Puis on propose un algorithme **exponentiel** en ce paramètre, mais **polynomial** si le paramètre est borné

Remarques

- Généralement plus **difficile** à implémenter
- Donne de bons résultats en **pratique**, pas juste en **théorie**
- Beaucoup de travaux de recherche **récents**

Définitions

Paramètre

- Un **paramètre** est une fonction qui associe à une instance un nombre naturel
- Par exemple, sa **taille**, la **taille** d'une solution optimale, le **nombre** de contraintes, ...

Résoluble avec paramètre fixé

- En anglais, *FPT = fixed parameter tractable*
- Soit k un paramètre
- Un problème est **résoluble avec paramètre fixé** k

$\rightarrow$ s'il existe une solution dont la **complexité** est dans $f(k)n^{\mathcal{O}(1)}$,

$\rightarrow$ où f est une fonction qui **ne dépend que** de k

Couverture des arêtes par les sommets

- Soit G un graphe de n sommets et m arêtes
- Soit k la taille d'une couverture optimale
- Alors on peut compter toutes les couvertures de taille au plus k en temps $\Theta(2^k(n+m))$
- Ainsi, si k est **petit** (*borné*), alors on résoud le problème en temps **polynomial**!

Idée-clé

- Soit $\{u, v\}$ une arête
- Alors il y a **deux possibilités**:

$\rightarrow$ u est dans la couverture

$\rightarrow$ et donc on peut supprimer ses arêtes incidentes

$\rightarrow$ **ou bien** v est dans la couverture

$\rightarrow$ et donc on peut supprimer ses arêtes incidentes

- Il ne reste qu'à **borner** la profondeur des appels récursifs par k

Pseudocode

```
1:  fonction VertexCover(G : graphe, C : couverture) : couverture
2:      si |C| = k et E(G) ≠ ∅ alors
3:          retourner ∅
4:      sinon si E(G) = ∅ alors
5:          retourner C
6:      Soit {u, v} une arête de G
7:      G' ← G − {u}
8:      C' ← C ∪ {u}
9:      si VertexCover(G', C') ≠ ∅ alors retourner C
10:     G' ← G − {v}
11:     C' ← C ∪ {v}
12:     si VertexCover(G', C') ≠ ∅ alors retourner C
```

Autres paramètres

Largeur arborescente

- Un paramètre **complexe** qui permet de classer les graphes
- Si la **largeur est 1**, on a un **arbre**
- Si la **largeur est 2**, on a un **graphe série-parallèle**
- Si la **largeur est 3 ou plus**, la classification est **difficile**

Largeur de chemin

- Un autre paramètre **similaire** à la largeur arborescente
- Indique à quel point un graphe est proche d'être un **chemin**

Intérêt?

Beaucoup de problèmes **NP-difficiles** connus sont résolubles avec paramètre fixé en prenant la **largeur arborescente** ou la **largeur de chemin**